The Multi-Principal OS Construction of the Gazelle Web Browser

Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, Herman Venter
Microsoft Research, University of Illinois at Urbana-Champaign, University of Washington
{helenw, pialic, hermanv}@microsoft.com, {grier, kingst}@uiuc.edu, anm@cs.washington.edu

Abstract

Web browsers originated as applications that people used to view static web sites sequentially. As web sites evolved into dynamic web applications composing content from various web sites, browsers have become multi-principal operating environments with resources shared among mutually distrusting web site *principals*. Nevertheless, no existing browsers, including new architectures like IE 8, Google Chrome, and OP, have a multi-principal operating system construction that gives a browser-based OS the *exclusive* control to manage the protection of all system resources among web site principals.

In this paper, we introduce Gazelle, a secure web browser constructed as a multi-principal OS. Gazelle's Browser Kernel is an operating system that exclusively manages resource protection and sharing across web site principals. This construction exposes intricate design issues that no previous work has identified, such as legacy protection of cross-origin script source, and cross-principal, cross-process display and events protection. We elaborate on these issues and provide comprehensive solutions.

Our prototype implementation and evaluation experience indicates that it is realistic to turn an existing browser into a multi-principal OS that yields significantly stronger security and robustness with acceptable performance and backward compatibility.

1 Introduction

Web browsers have evolved to be a multi-principal operating environment where a principal is a web site [38]. Similarly to a multi-principal OS, recent proposals [10, 11, 22, 38, 41] and browsers like IE 8 [29] and Firefox 3 [14] advocate and support abstractions for cross-principal communication (e.g., `PostMessage`) and protection (for frames) to web programmers. Nevertheless, no existing browsers, including new architectures like IE 8 [23], Google Chrome [32], and OP [20], have a multi-principal OS construction that gives a browser-based OS, typically called Browser Kernel, the *exclusive* control to manage the protection and fair-sharing of all system resources among browser principals.

In this paper, we present a multi-principal OS construction of a secure web browser, called Gazelle. Gazelle's Browser Kernel *exclusively* provides cross-principal protection and fair sharing of *all* system resources. In this paper, we focus on resource protection only.

Browser Kernel runs in a separate OS process, directly interacts with the underlying OS, and exposes a set of system calls for browser principals. We draw the isolation boundary across the existing browser principal[1] defined by the same-origin policy (SOP) [34], namely, the triple of <protocol, domain-name, port>, using sandboxed OS processes. The processes are sandboxed so that they cannot interact with the underlying system and must use system calls provided by Browser Kernel. Principals' processes can communicate with one another through Browser Kernel using inter-process communications. Unlike all existing browsers except OP, our Browser Kernel offers the same protection to plugin content as to standard web content.

[1] In this paper, we use "principal" and "origin" interchangeably.

Such a multi-principal OS construction for a browser brings significant security and reliability benefits to the overall browser system: the compromise or failure of a principal affects that principal alone, and not other principals or Browser Kernel.

Although our architecture may seem to be a straightforward application of multi-principal OS construction to the browser setting, it exposes intricate problems that didn't surface in previous work, including dealing with legacy protection for cross-origin script source, display protection, and resource allocations in the face of cross-principal web service composition common on today's web. We detail our solutions to the first two problems and leave resource allocation as future work.

In our browser design, we take the general stance that security (maintaining the multi-principal OS principles by having Browser Kernel exclusively manage the resource protection and sharing) comes before backward compatibility. We will not trade significant security risks for compatibility. Nevertheless, we will also not settle on a design that breaks many parts of the web to secure just a few sites. We present design rationales for such decisions throughout our design.

We have built an IE-based prototype that realizes Gazelle's multi-principal OS architecture and at the same time utilizes all the backward-compatible parsing, DOM management, and JavaScript interpretation that already exist in IE. Our prototype experience indicates that it is feasible to turn an existing browser into a multi-principal OS while leveraging its existing capabilities.

With our prototype, we successfully browsed 19 out of the 20 Alexa-reported [5], most popular sites that we tested. The performance of the prototype is acceptable, and a significant portion of the overhead comes from IE instrumentation, which can be eliminated in a production implementation. We gave a preliminary evaluation on backward compatibility of the policy enforced by Gazelle for the 100 Alexa-reported, most popular sites. By enforcing Gazelle's security policy, we modify the behavior of eight out of 100 sites.

For the rest of the paper, we first give an in-depth comparison with related browser architectures in Section 2. We then describe Gazelle's security model contrasted with existing browsers' policies in Section 3. In Section 4, we present our architecture, its design rationale, and how we treat the subtle issue of legacy protection for cross-origin script source. In Section 5, we elaborate on our problem statement and design for cross-principal, cross-process display protection. We give a security analysis including a vulnerability study in Section 6. We describe our implementation in Section 7. We measured the performance of our prototype and studied Gazelle's backward compatibility with popular sites. We detail our evaluation methodology and results in Section 8. Finally, we conclude and address future work in Section 9.

2 Related Work

In this section, we discuss related browser architectures and compare them with Gazelle.

In concurrent work [32], Reis et al detailed the various process models supported by Google Chrome: 1) monolithic process, 2) process-per-browsing-instance, 3) process-per-site-instance, and 4) process-per-site. A browsing instance contains all interconnected (or inter-referenced) windows including tabs, frames and subframes *despite* their origin. A site instance is a group of same-site pages within a browsing instance. A site is defined as a set of origins, as defined by SOP, that share a registry-controlled domain name, that is, *attackerAd.socialnet.com*, *alice.profiles.socialnet.com*, and *socialnet.com* share the same registry-controlled domain name *socialnet.com*, and are considered to be the same site or principal by Google Chrome. Furthermore, Reis et al [32] gave the caveats that Chrome's current implementation does *not* support strict site isolation in model 3 and 4: embedded principals, such as a nested `iframe` sourced at a different origin from the parent page, are placed in the same process as the parent page.

Process models 1 and 2 of Google Chrome are insecure since they don't provide memory or other resource protection across multiple principals in a monolithic process or browser instance. Model 4 doesn't provide failure containment across site instances [32]. Google Chrome's process-per-site-instance model is

the closest to Gazelle's two processes-per-principal-instance model, but with several crucial differences: 1) Chrome's principal is site (see above) while Gazelle's principal is the same as the SOP principal. Chrome's decision is to allow a site to set `document.domain` to a postfix domain (*ad.socialnet.com* set to *socialnet.com*). We argue in Section 3 that this practice has significant security risks. 2) A parent page's principal and its embedded principals co-exist in the same process in Google Chrome, whereas Gazelle places them into separate processes. Pursuing this design led us to new research challenges including cross-principal, cross-process display protection (Section 5). 3) Plugin content from different principals or sites share a plugin process in Google Chrome, but are placed into separate processes in Gazelle. 4) Chrome also relies on its rendering processes to correctly control network requests to enforce the same-origin policy. These differences indicate that in Chrome, cross-principal (or -site) protection takes place in its rendering processes and its plugin process, in addition to its Browser Kernel. In contrast, Gazelle's Browser Kernel functions as an OS, managing cross-principal protection on all resources, including networking and display.

The OP web browser [20] uses processes to isolate browser components (i.e., HTML engine, JavaScript interpreter, rendering engine) as well as pages of the same origin. OP allows any security model to be specified with such a framework. However, this flexibility comes with cost — intimate interactions between browser components, such as JavaScript interpreter and HTML engine, must use IPC and go through its browser kernel. When targeting a specific security model, such as that of existing browsers or Gazelle, the additional IPC cost does not add any benefits: isolating browser components within an instance of a web page provides no additional security protection. Furthermore, besides plugins, basic browser components are fate-shared in web page rendering; the failure of any one browser component results in most web pages not functioning properly; therefore, process isolation across these components doesn't provide any failure containment benefits either. Lastly, OP's browser kernel doesn't provide all the cross-principal protection needed as an OS: it delegates display protection to its processes.

IE 8 [23] uses OS processes to isolate tabs from one another. This granularity is insufficient since a user may browse multiple mutually distrusting sites in a single tab and a web page may contain an iframe with content from an untrusted site (e.g., ads).

Tahoma [9] uses virtual machines to completely isolate (its own definition of) web applications, disallowing any communications between the VMs. A web application is specified in a manifest file provided to the virtual machine manager and typically contains a suite of web sites of possibly different domains. Consequently, Tahoma doesn't provide protection to existing browser principals. In contrast, Gazelle's Browser Kernel protects browser principals first hand.

3 Security model

3.1 Background: security model in existing browsers

The access and protection model for various resources are inconsistent in today's browsers. These inconsistencies present significant hurdles for web programmers to build robust web services. In this section, we give a brief background on the relevant security policies in existing browsers. Michal Zalewski gives an excellent and the most complete description of existing browsers' security model to date [43].

Script. The same-origin policy (SOP) [34] is the central security policy on today's browsers. SOP governs how scripts access the HTML document tree and remote store. SOP defines the *origin* to be the triple of <protocol, domain-name, port>. Two documents from different origins cannot access each other's HTML documents using the Document Object Model (DOM), which is the platform and language neutral interface that allows scripts to dynamically access and update the content, structure and style of a document [12]. A script can access its document origin's remote data store using the `XMLHttpRequest` object, which issues an asynchronous HTTP request to the remote server [40]. (XMLHttpRequest is the cornerstone of the AJAX programming.) SOP requires a script to issue an XMLHttpRequest to only its enclosing page's origin. A

script executes as the principal of its enclosing page though its source code is not readable in a cross-origin fashion.

For example, an <iframe> with source *http://a.com* cannot access any HTML DOM elements from another <iframe> with source *http://b.com* and vice versa. *a.com*'s scripts (regardless of where the scripts are hosted) can issue XMLHttpRequests to only *a.com*. Furthermore, *http://a.com* and *https://a.com* are different origins because of the protocol difference.

Cookies. Cookies have a different security policy from that of script. For cookie access, the principal is defined to be the URL including the path, but without the protocol [17, 27]. For example, if the page *a.com/dir/1.html* creates a cookie, then that cookie is accessible to *a.com/dir/2.html* and other pages from that directory and its subdirectories, but is not accessible to *a.com/*. Furthermore, *https://a.com/* and *http://a.com/* share the cookie store unless a cookie is marked with a "secure" flag. Non-HTTPS sites may still set `secure` cookies in some implementations, just not read them back [43]. The browser ensures that a site can only set its own cookie and that a cookie is attached only to HTTP requests to that site. The path-based security policy for cookies doesn't play well with SOP for scripts: Scripts can gain access to all cookies belonging to a domain despite path restrictions because scripts can access all resources in a domain regardless of the paths.

Plugins. Current major browsers cannot enforce any security on plugins because plugins are allowed to interact with the local operating system directly. The plugin content is subject to the security policies implemented in the plugin software rather than the browser.

3.2 Gazelle's security model

Unified SOP across all resources. Gazelle's security model is centered around protecting principals from one another by separating their respective resources into hardware-isolated protection domains. Any sharing between two different principals must be explicit using cross-principal communication (or IPC) mediated by Browser Kernel.

We use the same principal as SOP, namely, the triple of <protocol, domain − name, port>. While it is tempting to have a more fine-grained principal, such as a path-based principal, we need to be concerned with co-existing with current browsers [38]: The protection boundary of a more fine-grained principal, such as a path-based principal would break down in existing browsers. It is unlikely that web programmers would write very different versions of the same service to accommodate different browsers; instead, they would forego the more fine-grained principal and have a single code base.

The resources that need to be protected across principals [38] are memory such as the DOM objects and script objects, persistent state such as cookies, display, and network communications. With a principal-driven approach, we have a consistent security policy across all these resources. This is unlike current browsers where the security policies vary for different resources. For example, cookie uses a different principal than that of DOM objects (see the above section); descendant navigation policy [6] [7] also cross the SOP principal boundary implicitly (more in Section 5.1). We support some backward compatibility when there are no perceived security risks, such as certain cross-principal cookie accesses (more later), through the explicit use of cross-principal communications.

We extend the same principal model to all content types except scripts and style sheets (Section 4): the elements created by <object>, <embed>, <img>, and certain types of <input>[2] are treated the same as an <iframe>; if the included content is from a different origin, the content belongs to the principal of that origin. This means that we *enforce* SOP on plugin content[3]. This is consistent with the existing movement in popular plugins like Adobe Flash Player [18]: Starting with Flash 7, Adobe Flash Player uses the exact domain match (as in SOP) rather than the earlier "superdomain" match (where *www.adobe.com*

[2] <input> can be used to include an image using a "src" attribute.

[3] OP [20] calls this plugin policy the *provider domain policy*.

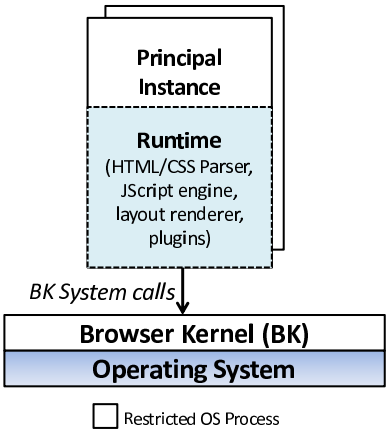

Figure 1: The basic architecture

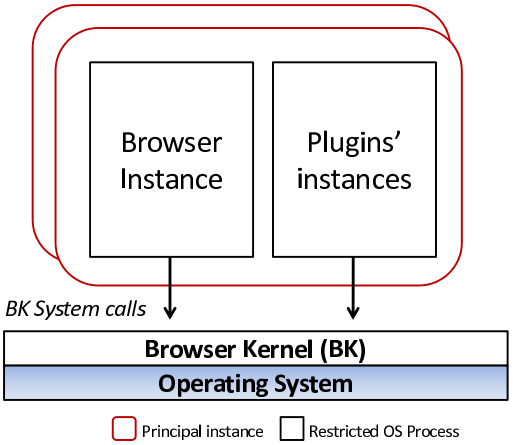

Figure 2: The Gazelle architecture

and *store.adobe.com* have the same origin) [2]; and starting with Flash 9, the default ActionScript behavior only allows access to same-origin HTML content unlike the earlier default that allows full cross-origin interactions [1].

Mixed HTTPS and HTTP Content. When an HTTPS site embeds HTTP content, browsers typically warn users about the mixed content since the HTTPS site's content can resist a network attacker, but the embedded HTTP content could be compromised by a network attacker.

When an HTTPS site embeds other HTTP principals (through `<iframe>`, `<object>`, etc.), HTTPS principals and HTTP principals will have different protection domains and won't interfere with each other.

However, when an HTTPS site embeds a script or style sheet delivered with HTTP, existing browsers would allow the script to run with the HTTPS site's privileges (after the user ignores the mixed content warning). This is dangerous because a network attacker can then compromise the HTTP-transmitted script and attack the HTTPS principal despite its intent of preventing network attackers. Therefore, in our system, we deny rendering HTTP-transmitted scripts or style sheets for an HTTPS principal.

Subdomain treatment. Existing browsers and SOP make exceptions for subdomains (e.g., *news.google.com* is a subdomain of *google.com*) [34]: a page can set the `document.domain` property to suffixes of its domain and assumes that identity. This feature was one of the few methods for cross-origin frames to communicate before the advent of `postMessage` [23]. Changing `document.domain` is a dangerous practice and violates the Principle of Least Privilege: Once a subdomain sets its domain to a suffix, it has no control over which other subdomains can access it. This is also observed by Zalewski [43]. Therefore, in Gazelle, we don't allow a subdomain to set `document.domain`.

Nevertheless, we do need to be concerned of backward compatibility when it comes to cookie access. Today, a subdomain can access a suffix domain's cookies, which is widely used for features, such as single sign-on. We support this through the use of cross-principal communications.

4 Architecture

4.1 Basic Architecture

Figure 1 shows our basic architecture. Different principals are placed into separate protection domains so that they are protected from one another. Just as in desktop applications where instances of an application

are run in separate processes for failure containment, we run instances of principals in separate protection domains for the same purpose. For example, when the user browses the same URL from different tabs, it corresponds to two instances of the same principal; when *a.com* embeds two *b.com* iframes, the *b.com* iframes correspond to two instances of *b.com*; however, multiple same-origin frames in a page are in the same principal instance as the page. Our principal instance is similar to Google Chrome's site instance [32], but with two crucial differences: 1) Google Chrome considers the sites that share the same registrar-controlled domain name to be from the same site, so *ad.datacenter.com*, *user.datacenter.com*, and *datacenter.com* are considered to be the same site and belong to the same principal. In contrast, we consider them as separate principals. 2) When a site, say *a.com*, embeds another principal's content, say an iframe with source *b.com*, Google Chrome puts them into the same site instance. In contrast, we put them into separate principal instances.

Browser Kernel runs in a separate protection domain and interposes between browser principals and the traditional OS and mediates their access to system resources. Essentially, Browser Kernel functions as an operating system to browser principals and manages the protection and sharing of system resources for them. Browser Kernel also manages the browser chrome, such as the address bar and menus. Browser Kernel receives all events generated by the underlying operating system including user events like mouse clicks or keyboard entries; then these events are dispatched to the appropriate principal instance. When the user navigates a window by clicking on a hyperlink which points to an URL at a different origin, Browser Kernel creates the protection domain for the URL's principal instance (if one doesn't exist already) to render the target page, destroys the protection domain of the hyperlink's host page, and re-allocates and re-initializes the window to the URL's principal instance. Browser Kernel is agnostic of DOM and content semantics and has a relatively simple logic.

The runtime of a principal instance is essentially an instance of today's browser components including HTML and style sheet parser, JavaScript engine, layout renderer, and browser plugins. The only way for a principal instance to interact with system resources, such as networking, persistent state, and display, is to use Browser Kernel's system calls. Principals can communicate with one another using message passing through Browser Kernel, in the same fashion as inter-process communications (IPC). It is necessary that the protection domain of a principal instance is a restricted or sandboxed OS process. The use of process guarantees the isolation of principals even in the face of attacks that exploit memory vulnerabilities. The process must be further restricted so that any interaction with system resources is limited to Browser Kernel system calls. Xax [13], Native Client [42], and GreenBorder [19] have established the feasibility of such process sandboxing.

This architecture is efficient. By putting all browser components including plugins into one process, they can interact with one another through DOM intimately and efficiently as they do in existing browsers. This is unlike the OP browser's approach [20] in which all browser components are separated into processes; chatty DOM interactions must be layered over IPCs through the OP browser kernel incurring unnecessary overhead without added security.

Unlike all existing browsers except OP, this architecture can enforce browser security policies on plugins, namely, plugin content from different origins are segregated into different processes. Any plugin installed is unable to interact with the operating system and is only provided access to system resources subject to Browser Kernel allowing that access. In this architecture, the payload that exploits plugin vulnerabilities will only compromise the principal with the same origin as the malicious plugin content, but not any other principals nor Browser Kernel.

Browser Kernel supports the following system calls related to content fetching in this architecture:

- *getSameOriginContent (URL)*: Fetch the content at URL that has the same origin as the issuing principal regardless of the content type.
- *getCrossOriginContent (URL)*: Fetch the script or style sheet content from URL; URL may be from

different origin than the issuing principal. The content type is determined by the `content − type` header of the HTTP response.

- *delegate (URL, windowSpec)*: Delegate a display area to a different principal of URL and fetch the content for that principal.

The semantics of these system calls is that Browser Kernel may return cross-origin script or style content to a principal based on the content-type header of the HTTP response, but returns other content if and only if the content has the same origin as the issuing principal, abiding the same-origin policy. All the security decisions are made and enforced by Browser Kernel alone.

4.2 Supporting Legacy Protection

The system call semantics in the basic architecture has one subtle issue: cross-origin script or style sheet sources are readable by the issuing principal, which does not conform with the existing SOP. The SOP dictates that a script can be executed in a cross-origin fashion, but the access to its source code is restricted to same-origin only.

A key question to answer is that whether a script should be processed in the protection domain of its provider (indicated in "src"), in the same way as frames, or in the protection domain of the host page that embeds the script. To answer this question, we must examine the primary intent of the script element abstraction. Script is primarily a *library* abstraction (which is a necessary and useful abstraction) for web programmers to include in their sites and run with the privilege of the includer sites [38]. This is in contrast with the frame abstractions: Programmers put content into cross-origin frames so that the content runs as the principal of its own provider and be protected from other principals. Therefore, a script should be handled by the protection domain of its includer.

In fact, it is a flaw of the existing SOP to offer protection for cross-origin script source. Evidences have shown that it is extremely dangerous to hide sensitive data inside a script [21]. Numerous browser vulnerabilities exist for failing to provide the protection.

Unfortunately, web sites that rely on cross-origin script source protection, exist today. For example, GMail's contact list is stored in a script file, at the time of writing. Furthermore, it is increasingly common for web programmers to adopt JavaScript Object Notation (JSON) [26] as the preferred data-interchange format. Web sites often demand such data to be same-origin access only. To prevent such data from being accidentally accessed through <`script`> (by a different origin), web programmers sometimes put "while (1);" prior the data definition or put comments around the data so that accidental script inclusion would result in infinite loop execution or no op.

In the light of the existing use, we modify our architecture slightly, as shown in Figure 2, to offer additional protection to cross-origin script source (though we do anticipate web programmers will move away from the dangerous practice of putting sensitive data in scripts).

Third-party plugin software vulnerabilities have surged recently [31]. Symantec reports that in 2007 alone there are 467 plugin vulnerabilities [37], which is about one magnitude higher than that of browser software. Clearly, plugin software should be trusted much less than browser software. Therefore, for protecting cross-origin script or style sheet source, we place more trust in the browser code and let browser code retrieve and protect cross-origin script or style sheet sources: For each principal, we run browser code and plugin code in two separate processes as shown in Figure 2. The plugin instance process cannot issue the `getCrossOriginContent()` and it can only interact with cross-origin scripts and style sheets through the browser instance process.

In this architecture, the quality of protecting cross-origin script and style-sheet source relies on the browser code quality. While this protection is not perfect with native browser code implementation, the architecture offers the same protection as OP, and stronger protection than the rest of existing browsers.

The separation of browser code and plugin code into separate processes also improves reliability by containing plugin failures.

Recent work Xax [13] and Native Client [42] have presented a plugin model that uses sandboxed process to contain each browser principal's plugin content. Their plugin model works perfectly in our browser architecture. We don't provide further discussions on plugins in our paper.

5 Cross-Principal, Cross-Process Display and Events Protection

Cross-principal service composition is a salient nature of the web and is commonly used in web applications. When building a browser as a multi-principal OS, this composition raises new challenges in display sharing and event dispatching: when a web site embeds a cross-origin frame (or objects, images), the involved principal instances share the display at the same time. Therefore, it is important that Browser Kernel 1) discerns display and events ownership, 2) enforces that a principal instance can only draw in its own display areas, 3) dispatches UI events to only the principal instance with which the user is interacting. An additional challenge is that Browser Kernel must accomplish these without access to any DOM semantics.

From a high level, in Gazelle we allow principal instances to render content into bitmap objects, and our Browser Kernel manages these bitmap objects and chooses when and where to display them. Our architecture provides a clean separation between the act of rendering web content and the policies of how to display this content. This architecture is a stark contrast to today's browsers that intermingle these two functions, which has lead to numerous security vulnerabilities [16, 39].

Our display management fundamentally differs from that of the traditional multi-user OSes, such as Unix and Windows. Traditional OSes offer no cross-principal display protection. In X, all the users who are authorized (through `.Xauthority`) to access the display can access one another's display and events. Experimental OSes like EROS [36] have dealt with cross-principal display protection. However, the browser context presents new challenges that don't exist in EROS, such as dual ownership of display and cross-principal transparent overlays.

5.1 Display Ownership and Access Control

We define *window* to be a unit of display allocation and delegation. Each window is allocated by a *landlord* principal instance or Browser Kernel; and each window is delegated to (or rented to) a *tenant* principal instance. For example, when the web site *a.com* embeds a frame sourced at *b.com*, *a.com* allocates a window from its own display area and delegates the window to *b.com*; *a.com* is the landlord of the newly-created window, while *b.com* is the tenant of that window. The same kind of delegation happens when cross-origin object and image elements are embedded. Browser Kernel allocates top-level windows (or tabs). When the user launches a site through address-bar entry, Browser Kernel delegates the top-level window to the site, making the site a tenant. We don't use "parent" and "child" terminologies because they only convey the window hierarchy, but not the principal instances involved. In contrast, "landlord" and "tenant" convey both semantics.

Window creation and delegation result in a `delegate`(`URL`, `position`, `dimensions`) system call. For each window, Browser Kernel maintains the following state: its landlord, tenant, position, dimensions, pixels in the window, and the URL location of the window content. Browser Kernel manages a three-dimensional display space where the position of a window also contains a stacking order value (toward the browsing user). A landlord provides the stacking order of all its delegated windows to Browser Kernel. The stacking order is calculated based on the DOM hierarchy and the CSS z-index values of the windows.

Because a window is created by a landlord and occupied by a tenant, Browser Kernel must allow reasonable window interactions from both principal instances without losing protection. Browser Kernel provides

	landlord	tenant
position (x,y,z)	RW	
dimensions (height, width)	RW	R
pixels		RW
URL location	W	RW

Table 1: Access control policy for a window's landlord and tenant

the access control as follows:

- *Position and dimensions*: When a landlord embeds a tenant's content, the landlord should be able to retain control on what gets displayed on the landlord's display and a tenant should not be able to reposition or resize the window to interfere with the landlord's display. Therefore, Browser Kernel enforces that only the landlord of a window can change the position and the dimensions of a window.
- *Drawing isolation*: Pixels inside the window reflect the tenant's private content and should not be accessible to the landlord. Therefore, Browser Kernel enforces that only the tenant can draw within the window. (Nevertheless, a landlord can create overlapping windows delegated to different principal instances.)
- *Navigation*: Setting the URL location of a window navigates the window to a new site. Navigation is a fundamental element of any web application. Therefore, both the landlord and the tenant are allowed to set the URL location of the window. However, the landlord should not obtain the tenant's navigation history that is private to the tenant. Therefore, Browser Kernel prevents the landlord from reading the URL location. The tenant can read the URL location as long as it remains being the tenant. (When the window is navigated to a different principal, the old tenant will no longer be associated with the window and will not be able to access the window's state.)

Table 1 summarizes the access control policies in Browser Kernel. In existing browsers, these manipulation policies also vaguely exist. However, their logic is intermingled with the DOM logic and is implemented at the object property and method level of a number of DOM objects which all reside in the same protection domain despite their origins. This had led to numerous vulnerabilities [16, 39]. In Gazelle, by separating these security policies from the DOM semantics and implementation, and concentrating them inside Browser Kernel, we achieve much more clarity in our policies and much stronger robustness in the system construction.

Browser Kernel ensures that principal instances other than the landlord and the tenant cannot manipulate any of the window states. This includes manipulating the URL location for navigation. Here, we depart from the existing descendant navigation policy in most of today's browsers [6, 7]. Descendant navigation policy allows a landlord to navigate a window created by its tenant even if the landlord and the tenant are different principals. This is flawed in that a tenant-created window is a resource that belongs to the tenant and should not be controllable by a different principal.

Existing literature [6, 7] supports the descendant navigation policy with the following argument: since existing browsers allow the landlord to draw over the tenant, a landlord can simulate the descendant navigation by overdrawing. Though overdrawing can *visually* simulate navigation, navigation is much more powerful than overdrawing because a landlord with such descendant navigation capability can interfere with the tenant's operations. For example, a tenant may have a script interacting with one of its windows and then effecting changes to the tenant's backend; navigating the tenant's window requires just one line of JavaScript and could effect undesirable changes in the tenant's backend. With overdrawing, a landlord can imitate a tenant's content, but the landlord cannot send messages to the tenant's backend in the name of the tenant.

5.2 Cross-Principal Events Protection

Browser Kernel captures all events in the system and must accurately dispatch them to the right principal instance to achieve cross-principal event protection. Networking and persistent-state events are easy to dispatch. However, user interface events pose interesting challenges to Browser Kernel in discerning event ownership, especially when dealing with overlapping, potentially transparent cross-origin windows: major browsers allow web pages to mix content from different origins along the z axis where content can be occluded, either partially or completely, by cross-origin content. In addition, current standards allow web pages to make a frame or portions of their windows transparent, further blurring the lines between principals. Although these flexible mechanisms have a slew of legitimate uses, they can be used to fool users into thinking they are interacting with content from one origin, but are in fact interacting with content from a different origin. For example, recent "UI redressing" attacks [43] illustrate some of the difficulties with current standards and how attackers can abuse these mechanisms.

To achieve cross-principal event protection, Browser Kernel needs to determine the *event owner*, the principal instance to which the event is dispatched. There are two types of events: stateless and stateful. The owner of a stateless event like a mouse event is the tenant of the window (or display area) on which the event takes place. The owner of a stateful event such as a key-press event is the tenant of the current in-focus window. Browser Kernel interprets mouse clicks as focus-setting events and keeps track of the current in-focus window and its principal instance.

The key problem to solve then is to determine the window on which a stateless or focus-setting event takes place. We consider a determination to have high *fidelity* if the determined event owner corresponds to the user intent. Different window layout policies directly affect the fidelity of this determination. We explore various layout policies and settle on the policy that gives us the best fidelity, security, and backward compatibility.

Existing browsers' policy. The layout policy in existing browsers is to draw windows according to the DOM hierarchy and the z-index values of the windows. Existing browsers then associate a stateless or focus-setting event to the window that has the highest stacking order. Today, most browsers permit page authors to set transparency on cross-origin windows [43]. This ability can result in poor fidelity in determining event owner in the face of cross-principal transparent overlays: When there are transparent, cross-origin windows overlapping with one another, it is impossible for Browser Kernel to interpret the user's intent: the user is guided by what she sees on the screen; when two windows present a mixed view, some user interfaces visible to the user belong to one window, and yet some belong to another. For two overlapping, cross-origin windows, such mixing is often not mutually intended by page authors as well (otherwise, they would be same-origin windows). The ability to overlay transparent cross-origin content is also extremely dangerous: a malicious site can make an iframe sourced at a legitimate site tranparent and overlaid on top of the malicious site [43], fooling the users to interact with the legitimate site unintentionally.

The two layout policies below pose additional constraints on layout policies so that the user intent and event owners can be determined more reliably and securely, but they have different trade-offs between security and backward compatibility.

2-D display delegation policy[4]. In this policy, the display is managed as two-dimensional space for the purpose of delegation. Once a landlord delegates a rectangular area to a tenant, the landlord cannot overdraw the area. Thus, no cross-principal content can be overlaid. Such a layout constraint will enable perfect fidelity in determining an event ownership that corresponds to the user intent. It also yields better security as it can prevent all UI redressing attacks except clickjacking[5] [43].

However, this policy can have a significant impact on backward compatibility. For example, a menu from

[4]We called this policy "pixel isolation" in a previous version of this paper.

[5]Clickjacking would be extremely difficult to launch with this policy on our system since our cross-principal memory protection makes reading and writing the scrolling state of a window an exclusive right of the tenant of the window.

a host page cannot be drawn over a nested cross-origin frame or object; many sites would have significant constraints with their own DOM-based pop-up windows created with `divs` and such (rather than using `window.open` or `alert`), which could overlay on cross-origin frames or objects with existing browsers' policy; a cross-origin image cannot be used as a site's background.

Gazelle's opaque overlay policy. In Gazelle, we retain existing browsers' display management and layout policies as much as possible for backward compatibility (and additionally provide cross-principal events protection), but add the following layout constraint: Browser Kernel enforces that *each dynamic content-containing window, namely frames and objects, must be opaque*. We allow a page to use a cross-origin image (which are static content) as its background. Note that no principal instance other than the tenant of the window can set the background of a window due to our memory protection across principal instances. So, it is impossible for a principal to fool the user by setting another principal's background. Browser Kernel associates a stateless event or a focus-setting event with the dynamic content-containing window that has the highest stacking order.

This policy prevents an attack page from overlaying a transparent victim page over itself. However, by allowing overlapping opaque cross-origin frames or objects, our policy allows not only legitimate uses, such as those denied by the 2D display delegation policy, but it also allows an attacker page to cover up and expose selective areas of a nested cross-origin victim frame or object. This policy can result in infidelity for the latter scenario. A potential mitigation is for Browser Kernel to determine how much of a page's content is exposed in an undisturbed fashion to the user when the user clicked on the page. We are exploring such a mitigation strategy and a heuristic that yields no or extremely low false positives as well as low false negatives.

One additional consideration is race condition attacks [43]. In such an attack, a malicious page may predict a user click (based on earlier user behaviors) and then exposes a screen area just before the click takes place, making the user unintentionally click on the newly exposed screen area. The 2D display delegation policy above eliminates this attack because of no overlapping of cross-origin windows at all. With the opaque transparency policy, we mitigate this category of attacks as follows: Browser Kernel keeps track of newly exposed screen areas (caused by window creation, repositioning or resizing). It ignores any click events at the newly exposed screen area until the user has had enough time (1 second) to see the area. This solution will mitigate scenarios where users click on a legitimate site (shown by an attacker site) unintentionally. It is not meant to prevent picture-in-picture attacks where users are fooled and consciously click on attacker sites.

We chose the opaque overlay policy together with the above mitigation mechanism as our design because it presents the best tradeoff for fidelity, security, and backward compatibility. This design yields much better fidelity and security than that of existing browsers though it doesn't offer as perfect fidelity and as strong resilience to UI redressing attacks as does the 2D display delegation policy. Nevertheless, this design offers significantly better backward compatibility than the 2D display delegation policy.

6 Security Analysis

In Gazelle, the trusted computing base encompasses Browser Kernel and the underlying OS. If Browser Kernel is compromised, the entire browser is compromised. If the underlying OS is compromised, the entire host system is compromised. If the DNS is compromised, all the non-HTTPS principals can be compromised. When Browser Kernel, DNS, and the OS are intact, our architecture guarantees that the compromise of a principal instance does not give it any capabilities additional to those already granted to it through Browser Kernel system call interface (Section 4).

Next, we analyze Gazelle's security over classes of browser vulnerabilities. We also make a comparison with popular browsers with a study on their past, known vulnerabilities.

	IE 7	Firefox 2
origin validation error	6	11
memory error	38	25
GUI logic flaw	3	13
others	-	28
total	47	77

Table 2: Vulnerability Study for IE 7 and Firefox 3

- Cross-origin vulnerabilities: By separating principals into different protection domains and making any sharing explicit, we can much more easily eliminate cross-origin vulnerabilities. The only logic for which we need to ensure correctness is the origin determination in Browser Kernel.

  This is unlike existing browsers, where origin validations and SOP enforcement are spread through the browser code base [8] and content from different principals co-exist in shared memory. All of the cross-origin vulnerabilities illustrated in Chen et al. [8] simply do not exist in our system; it does *not* require any special logic to prevent them because all those vulnerabilities exploit implicit sharing.

- Display vulnerabilities:

  The display is also a resource that Gazelle's Browser Kernel protects across principals, unlike existing browsers (Section 5). Cross-principal display and events protection and access control are enforced in Browser Kernel. This prevents a potentially compromised principal from hijacking the display and events that belong to another principal. Display hijacking vulnerabilities have manifested in existing browsers [15, 24] that allow an attacker site to control another site's window content.

  Gazelle's layout and event dispatching policy (Section 5.2) prevents transparency on cross-origin frames and plugins and can successfully prevent a malicious site from overlaying a transparent victim site. These attacks manifest in existing browsers when an attacker places a decoy user interface underneath malicious UI components with transparent CSS settings.

  Race condition attacks [43] are caused by an attacker predicting the time and location of user input and presents the victim site's user interface at the location of user input. Gazelle prevents these attacks by ignoring user input into newly exposed window area (for one second) until the user has got a chance to see the newly exposed display.

- Plugin vulnerabilities:

  Third-party plugins have emerged to be a significant source of vulnerabilities [31]. Unlike existing browsers, plugins in our system can only interact with system resources by means of Browser Kernel system calls so that they are subject to our browser's security policy. Plugins are contained inside sandboxed processes so that basic browser code doesn't fate-share with plugin code (Section 4). A compromised plugin affects the principal instance's plugin process only, and not other principal instances nor the rest of the system. In contrast, in existing browsers, a compromised plugin undermines the entire browser and often the host system as well.

  A DNS rebinding attack results in the browser labeling resources from different network hosts with a common origin. This allows an attacker to operate within SOP and access unauthorized resources [25]. Although Gazelle does not fundamentally address this vulnerability, the fact that plugins must interact with the network through Browser Kernel system calls defeats the multipin form of such attacks.

We analyzed the known vulnerabilities of two major browsers, Firefox 2 [3] and IE 7 [30], as shown in Table 2. For both browsers, memory errors are a significant source of errors. Memory-related vulnerabilities

Type	Call Name	Description
syscall	getSameOriginContent(URL)	retrieves same origin content
syscall	getCrossOriginContent(URL)	retrieves script or css content
syscall	delegate(URL, delegatedWindowSpec)	delegates screen area to a different principal
syscall	postMessage(windowID, msg, targetOrigin)	cross-frame messaging
syscall	display(windowID, bitmap)	sets the display buffer for the window
syscall	back()	steps back in the window history
syscall	forward()	steps forward in the window history
syscall	navigate (windowID, URL)	navigates a window to URL
syscall	createTopLevelWindow (URL)	creates a new browser tab for the URL specified
syscall	changeWindow (windowID, position, size)	updates the location and size of a window
syscall	writePersistentState (type, state)	allows writing to origin partitioned storage
syscall	readPersistentState (type)	allows reading of origin partitioned storage
syscall	lockPersistentState (type)	locks one type of origin partitioned storage
upcall	destroy(windowID)	closes a browser instance
upcall	resize(windowID, windowSpec)	changes the dimensions of the browser instance
upcall	createPlugin(windowID, URL, content)	creates a plugin instance
upcall	createDocument(windowID, URL, content)	creates a browser instance
upcall	sendEvent(windowID, eventInfo)	passes an event to the browser instance

Table 3: The Gazelle System Calls

are often exploited by maliciously crafted web pages to compromise the entire browser and often the host machines. In Gazelle, Browser Kernel is implemented with managed C# code and is resilient to memory attacks. Memory attacks in principal instances are well-contained in their respective sandboxed processes. Cross-origin vulnerabilities, or origin validation errors represent a significant portion of total vulnerabilities. These vulnerabilities result from the implicit sharing across principals in existing browsers and can be much more easily eliminated in our browser because cross-principal protection is exclusively handled by Browser Kernel and the use of sandboxed process. In IE 7, there are 3 GUI logic flaws which can be exploited to spoof the contents of the address bar. For Gazelle, the address bar UI is owned and controlled by our Browser Kernel, where it is much easier to implement correctly. In addition, Firefox has other errors which didn't map into these three categories, such as JavaScript privilege escalation, URL handling errors, and parsing problems. Since Gazelle enforces security properties in Browser Kernel, any errors that manifest as the result of JavaScript handling and parsing are limited in the scope of exploit to the principal instance owning the page. URL handling errors could occur in our Browser Kernel as well.

7 Implementation

We have built a Gazelle prototype as described in Section 4, running on Windows Vista with .NET framework 3.5 [4]. We next discuss the implementation of two major components shown in Figure 2: Browser Kernel the browser instance.

Browser Kernel. Browser Kernel consists of approximately 5k lines of C# code. It communicates with principal instances using system calls and upcalls, which are implemented as asynchronous XML-based messages sent over named pipes. An overview of Browser Kernel system calls and upcalls are listed in Table 3. Syscalls are performed by the browser instance or plugins and sometimes include replies. Upcalls are messages from Browser Kernel to the browser instance.

Display management is implemented as described in Section 5 using .NET's Graphics and Bitmap li-

braries. Each browser instance provides Browser Kernel with a bitmap for each window of its rendered content using a `display` system call; each change in rendered content results in a subsequent `display` call. For each top-level browsing window (or tab), Browser Kernel maintains a stacking order and uses it to compose various bitmaps belonging to a tab into a single master bitmap, which is then attached to the tab's PictureBox form. This straightforward display implementation has numerous optimization opportunities, many of which have been thoroughly studied [28, 33, 35], and which are not the focus of our work.

Browser instance. Instead of undertaking a significant effort of writing our own HTML parser, renderer, and JavaScript engine, we borrow these components from Internet Explorer 7 in a way that does not compromise security. Relying on IE's Trident renderer has a big benefit of inheriting IE's page rendering compatibility and performance. In addition, such an implementation shows that it is realistic to adapt an existing browser to use Gazelle's secure architecture.

In our implementation, each browser instance embeds a Trident `WebBrowser` control wrapped with an *interposition layer* which enforces Gazelle's security properties. The interposition layer uses Trident's COM interfaces, such as `IWebBrowser2` or `IWebBrowserEvents2`, to hook sensitive operations, such as navigation, frame creation, or image fetching, and convert them into system calls to the browser kernel. Likewise, the interposition layer receives browser kernel's upcalls, such as keyboard or mouse events, and synthesizes them in the Trident instance.

For example, suppose a user navigates to a web page `a.com`, which embeds a cross-principal frame `b.com`. First, Browser Kernel will fetch `a.com`'s HTML content, create a new `a.com` process with a Trident component, and pass the HTML to Trident for rendering. During the rendering process, we intercept the frame navigation event for `b.com`, determine that it is cross-principal, and cancel it. The frame's DOM element in `a.com`'s DOM is left intact as a placeholder, making the interposition transparent to `a.com`. We extract the frame's position, dimensions, and CSS properties from this element through DOM-related COM interfaces, and send this information in a `delegate` system call to Browser KernelḂrowser Kernel then creates a new `b.com` process (with a new instance of Trident), and asks it to render `b.com`'s frame. For any rendered display updates for either `a.com` or `b.com`, our interposition code obtains a bitmap of display content from Trident using the `IViewObject` interface and sends it to Browser Kernel for rendering.

One intricacy we faced was that passing raw HTML to a Trident renderer breaks any relative links on the page, since Trident forces injected HTML to have an `about:blank` URL. As a workaround, our interposition layer transparently rewrites web pages to include a <`base`> HTML element, which establishes the correct base path for any relative links.

Our interposition layer ensures that our Trident components are never trusted with sensitive operations, such as network access or display rendering. However, if a Trident renderer is compromised, it could bypass our interposition hooks and compromise other principals using the underlying OS's APIs. To prevent this, we are in the process of implementing an OS-level sandboxing mechanism, which would prevent Trident from directly accessing sensitive OS APIs. The feasibility of such a browser sandbox has already been established in prior work, such as Native Client [42], Xax [13] or GreenBorder [19]; our sandboxing mechanism is a straightforward engineering effort that builds on this work.

To verify that such an implementation does not cause rendering problems with popular web content, we used our prototype to manually browse through the top 20 Alexa [5] web sites. We checked the correctness of Gazelle's visual output against unmodified IE and briefly verified page interactivity, for example by clicking on links. We found that 19 of 20 web sites rendered correctly. The remaining web site exposed a (fixable) bug in our interposition code, which caused it to load with incorrect layout. Two sites experienced crashes (due to more bugs) when trying to render embedded cross-principal <`iframe`>'s hosting ads. However, the crashes only affected the <`iframe`> processes; the main pages rendered correctly with the exception of small blank spaces in place of the failed <`iframe`>'s. This illustrates a desirable security property of our architecture, which prevents malicious or misbehaving cross-origin tenants from affecting their landlords or other principals.

	Gazelle		Internet Explorer 7	
	Time	Memory Used	Time	Memory Used
1. Browser startup (no page)	225 ms	9 MB	635 ms	14 MB
2. New tab (blank page)	631 ms	14 MB	115 ms	0.7 MB
3. New tab (google.com)	970 ms	16 MB	499 ms	1.4 MB
4. Navigate from google.com to google.com/ads	811 ms	6 MB	1139 ms	3.1 MB
5. Navigate to nytimes.com (with a cross-origin frame)	6288 ms	88 MB	3213 ms	53 MB

Table 4: Loading times and memory overhead for a sequence of typical browser operations.

8 Evaluation

In this section, we evaluate the performance of the Gazelle prototype and the compatibility of Gazelle with a preliminary quantative study of popular web sites.

8.1 Performance

We measure the impact of our architecture on browser performance. All tests were performed on an Intel 2.66Ghz Core Duo with 2GB of memory and a 500GB SATA hard drive, running 32-bit Windows Vista with a gigabit Ethernet connection. To evaluate Gazelle's performance, we measured page load latencies, the memory footprint, and responsiveness of our prototype. We found that while Gazelle performs on-par with desktop browsers while browsing within an origin, it introduces some overhead for cross-origin navigation, e.g., 471 ms for `google.com` and 3 seconds for `NYTimes.com`. The overhead mainly stems from our Trident interposition layer, various initialization costs for new browser instances, and the unoptimized nature of our prototype. We point out simple optimizations that would mitigate much of the overhead along the way.

Page load latency. Table 4 shows the loading times for a series of browser operations a typical user might perform, using both our prototype and standalone Internet Explorer 7 which has a monolithic process architecture. The operations are repeated one after another within the same browser. A web page's loading time is defined as the time between pressing the "Go" button and seeing the fully-rendered web page. All operations include network latency.

Operation 1 measures the time to launch the browser; compared to IE7, Gazelle's Browser Kernel is faster to start and initialize because our Browser Kernel and its UI is much smaller than the monolithic IE7. Operations 2 and 3 each cause a new process to be created in Gazelle; compared to IE7, these operations incur additional process creation costs. Operation 4 reuses the same `google.com` process in Gazelle to render a same-origin page to which the user navigates via a link on `google.com`. Here, Gazelle actually outperforms IE, which is most likely due to IE's managing additional state such as browsing history between navigations. Finally, operation 5 causes Gazelle to create a new process for `nytimes.com` to render the popular news page. In addition, NYTimes contains an embedded cross-principal `<iframe>`, which triggers window delegation and another process creation event. The overall page load latency of 6288 ms includes the rendering times of both the main page and the embedded `<iframe>`, with the main page becoming visible and interactive to the user in 4918 ms.

It is expected that for most operations Gazelle will have a performance overhead as compared to standalone IE, due to process creation costs, messaging overhead, and the overhead of our Trident interposition layer as well as Trident itself. To better understand the overheads involved, Table 5 breaks down the total page loading times for the three sites of Table 4 across various code paths in the BK and browser instance processes.

		Latency		
Location	**Overhead**	blank site	google.com	nytimes.com
Browser kernel	Process creation	41 ms	54 ms	120 ms
Browser instance	Creating an interposed instance of Trident	102 ms	104 ms	210 ms
Browser instance	Named pipe initialization	129 ms	133 ms	533 ms
Browser kernel	Network access	6 ms	261 ms	448 ms
Browser instance	Trident parsing, rendering, and JS execution	106 ms	138 ms	2249 ms
Browser instance	Trident interposition overhead	74 ms	92 ms	1905 ms
Browser instance	Bitmap capturing	10 ms	14 ms	158 ms
Browser instance	Bitmap transfer	50 ms	63 ms	239 ms
Browser kernel	BK display rendering	34 ms	26 ms	78 ms
Browser kernel	Other BK code	13 ms	16 ms	31 ms
Browser instance	Other browser instance code	32 ms	28 ms	194 ms
Shared	Transferring system calls	33 ms	41 ms	123 ms
	Total	631 ms	969 ms	6288 ms

Table 5: A breakdown of overheads for page rendering times. Note that nytimes.com creates *two* processes for itself and an <iframe>; the other two sites create one process.

Our Trident interposition layer is a big source of overhead, especially for larger sites like NYTimes.com, where it consumes 1905 ms. Although we plan to optimize our use of Trident's COM interfaces, we are also limited by the Trident host's implementation of the hooks we rely on, and by the COM layer which exposes these hooks. Nevertheless, we believe we could mitigage most of this latency if Trident were to provide us with a direct (non-COM) implementation for a small subset of its hooks that Gazelle requires.

Process creation is an expected source of overhead that increases whenever sites embed cross-principal content, such as NYTimes's <iframe>. As well, each process must instantiate and initialize a new Trident object, which is expensive. As an optimization, we could use a worker pool of a few processes that have been pre-initialized with Trident. This would save us 330 ms on NYTimes's load time and 158 ms on `google.com`'s load time.

Gazelle does not currently overlap process creation with BK's network transfer, instead waiting for a new browser instance to initialize and make a *getSameOriginContent* system call. In the case of a user-typed URL, the BK could predict such a call and start pre-fetching the URL while the browser instance processes are being initialized, effectively masking much of network latency. Moreover, the BK currently releases web page data only when the network transfer finishes; instead, the BK could provide browser instances with chunks of data as soon as they arrive (e.g., by changing `getContent` system calls to the semantics of a UNIX `read()` system call), overlapping large network transfers with rendering.

We encountered an unexpected performance hit when initializing named pipes that we use to transfer system calls: a new process's first write to a pipe stalls for a considerable time. This could be caused by initialization of an Interop layer between .NET and the native Win32 pipe interfaces, on which our implementation relies. We can avoid this overhead by either using an alternate implementation of a system call transfer mechanism, or pre-initializing named pipes in our worker pool. This would save us 533 ms in NYTimes's render time.

Retrieving bitmap display updates from Trident and sending them to BK is expensive for large, complex sites such as NYTimes.com, where this takes 397 ms. Numerous optimizations are possible, including image compression, VNC-like selective transfers, and a more efficient bitmap sharing channel between Trident and BK. Our mechanism for transferring bitmap updates currently performs an inefficient .NET-based serialization of the image's data (which takes 176 ms for NYTimes); passing this data directly would

further improve performance.

Memory overhead. As a baseline measurement, Browser Kernel occupies around 9MB of memory after a page load. This includes the user interface components of the browser to present the rendered page to the user and the buffers allocated for displaying the rendered page. Memory measurements do not include shared libraries used by multiple processes.

Table 4 shows the amount of memory for performing various browsing operations. For example, to open a new tab to a blank page, Gazelle consumes 14MB, and to open a new tab for `google.com`, Gazelle consumes an additional 16MB. Each empty browser instance uses 1.5MB of internal storage plus the memory required for rendered content. Given our implementation, the latter closely corresponds to Trident's memory footprint, which at the minimum consists of 14MB for a blank page. In the case of NYTimes, our memory footprint further increases because of structures allocated by the interposition layer, such as a local DOM cache.

Responsiveness. We evaluated the response time of a user-generated event, such as a mouse click. When the BK detects a user event, it issues a *sendEvent* upcall to the destination principal's browser instance. Such calls take only 2 ms on average to transfer, plus 1 ms to synthesize in Trident. User actions might lead to display updates; for example, a display update for `google.com` would incur an additional 77 ms. Most users should not perceive this overhead and will experience good responsiveness.

Process creation. In addition to latency and memory measurements we also have tested our prototype on the top 100 popular sites reported by Alexa [5] to provide an estimate of the number of processes created for different sites. The number of processes created is determined by the use of different-origin content on sites, which is most commonly image content. For the top 100 sites, the median number of processes required to view a single page is 4, the minimum is 1, and the maximum is 28 (caused by `skyrock.com`, which uses an image farm). Although creation of many processes introduces additional latency and memory footprint, we did not experience difficulties when Gazelle created many processes during normal browsing. Our test machine easily handles a hundred running processes, which are enough to keep 25 average web sites open simultaneously.

8.2 Compatibility

In this section, we give a preliminary compatibility study on Gazelle over the 100 most popular web sites ranked by Alexa [5]. We use an instrumented browser to fetch content and execute JavaScript automatically. For our quantitative comparisons, we consider any visual differences in the rendering of a web page to be violation of compatibility. This conservative metric helps to give us an upper bound on the number of sites from the top 100 that are incompatible with Gazelle.

Plugins. Gazelle enforces the same-origin policy on plugin content. That is, plugin content runs as the principal of its origin (and not as the principal of its host page). Since plugins are a source of rampant vulnerability growth, enforcing policy on plugins in Browser Kernel is critical for containing the damage caused by an exploit. Existing plugin software must be adapted (ported or binary-rewritten) to use Browser Kernel system calls to accomplish its tasks. There are 34 out of top 100 Alexa sites that use Flash plugins and no other kinds of plugins are used. This indicates that by porting or adapting Flash alone can address a significant portion of the plugin compatibility issue.

document.domain. Gazelle prohibits modification of `document.domain` for changing a document's principal. Most current browsers allow document.domain to be set to suffixes of the document's domain, though they prevent the top and second-level domains from being changed. Our experiments indicate that six of the top 100 Alexa sites set `document.domain` to a different origin, though restricting write access to `document.domain` might not actually break the operation of these web sites.

Although we currently prevent this behavior in Gazelle and favor `postMessage` for cross-window communication, Gazelle could utilize a migration strategy to change the principal for a page by moving the page

from one principal to another.

HTTPS inclusion of HTTP scripts and CSS. In Gazelle an HTTPS principal cannot include an HTTP script or CSS content. Other types of content, such as images and plugins, will be run in the process belonging to the hosting principal. This policy could be easily changed in Gazelle; however, by preventing HTTPS content from including HTTP provided scripts and CSS, Gazelle enforces stronger security against a network attacker. Since the top 100 Alexa sites don't use SSL, we identified a few different sites that provide SSL sessions for parts of their web application: `amazon.com`, `mail.google.com`, `mail.microsoft.com`, `blogger.com`, and a few popular banking sites that are outside of the top 100. These sites don't violate our policy.

Strict child-only navigation. A landlord frame can only cause navigation in tenants and has no control over frames belonging to the tenants or other principals. Sites that attempt to navigate descendants other than direct children will be unable to do so in Gazelle. By limiting the scope of navigation, Gazelle is able to prevent malicious web pages from navigating frames in benign web applications. This type of attack was previously analyzed by Barth *et al.* [7] and presented as the "recursive mashup attack". We were unable to automatically test sites for violations of this policy with our compatibility testing framework.

Layering of different-origin content. Gazelle's opaque overlay policy allowing only opaque (and not transparent) frames or objects (Section 5.2), unlike existing browsers. We test this policy with the top 100 Alexa sites by determining if any cross-origin frames or objects are overlapped with one another. We found that two out of 100 sites attempt to violate this policy. This policy does not generate rendering errors; instead, we convert transparent cross-origin elements to opaque elements when displaying content.

We also tested the 2D display delegation policy that we analyzed in Section 5.2. We found this policy to have higher compatibility cost than our opaque overlay policy: six of the top 100 sites attempt to violate this policy.

9 Concluding Remarks

We have presented Gazelle, the first web browser that qualifies as a multi-principal OS for web site principals. This is because Gazelle's Browser Kernel *exclusively* manages resource protection, unlike all existing browsers which allow cross-principal protection logic to reside in the principal space. Gazelle enjoys the security and robustness benefit of a multi-principal OS: a compromise or failure of one principal leaves other principals and the Browser Kernel intact.

Our browser construction exposes challenging design issues that were not seen in previous work, such as providing legacy protection to cross-origin script source and cross-principal, cross-process display and event protection. We are the first to provide comprehensive solutions to them.

The implementation and evaluation of our IE-based prototype shows promise of a practical multi-principal OS-based browser in the real world. In our future work, we are exploring the fair sharing of resources among web site principals in our Browser Kernel.

References

[1] Changes in allowScriptAccess default (Flash Player). `http://www.adobe.com/go/kb403183`.

[2] Developer center: Security changes in flash player 7. `http://www.adobe.com/devnet/flash/articles/fplayer_security.html`.

[3] Security advisories for firefox 2.0. `http://www.mozilla.org/security/known-vulnerabilities/firefox20.html`.

[4] .NET Framework Developer Center, 2008. `http://msdn.microsoft.com/en-us/netframework/default.aspx`.

[5] Alexa, 2009. `http://www.alexa.com/`.

[6] A. Barth and C. Jackson. Protecting browsers from frame hijacking attacks, April 2008. http://crypto.stanford.edu/websec/frames/navigation/.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Usenix Security*, August 2008.

[8] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[9] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[10] D. Crockford. JSONRequest. http://www.json.org/jsonrequest.html.

[11] D. Crockford. The Module Tag: A Proposed Solution to the Mashup Security Problem. http://www.json.org/module.html.

[12] Document Object Model. http://www.w3.org/DOM/.

[13] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2008.

[14] Firefox 3 for developers, 2008. https://developer.mozilla.org/en/Firefox_3_for_developers.

[15] Mozilla Browser and Mozilla Firefox Remote Window Hijacking Vulnerability, 2004. http://www.securityfocus.com/bid/11854/.

[16] Security Advisories for Firefox 2.0. http://www.mozilla.org/security/known-vulnerabilities/firefox20.html.

[17] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media Inc., August 2006.

[18] Adobe Flash Player 9 Security, july 2008. http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf.

[19] Greenborder technologies. http://en.wikipedia.org/wiki/GreenBorder.

[20] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Securiy and Privacy*, 2008.

[21] J. Grossman.
Advanced Web Attack Techniques using GMail. http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html.

[22] W. H. A. T. W. Group. Web Applications 1.0, February 2007. http://www.whatwg.org/specs/web-apps/current-work/.

[23] What's New in Internet Explorer 8, 2008. http://msdn.microsoft.com/en-us/library/cc288472.aspx.

[24] Microsoft Internet Explorer Remote Window Hijacking Vulnerability, 2004. http://www.securityfocus.com/bid/11855.

[25] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *Proceedings of ACM Conference on Computer and Communications Security*, 2007.

[26] JavaScript Object Notation (JSON). http://www.json.org/.

[27] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2965, October 2000.

[28] T. W. Mathers and S. P. Genoway. *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*. Macmillan Technical Publishing, Indianapolis, IN, November 1998.

[29] IEBlog: IE8 Security Part V: Comprehensive Protection, 2008. http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx.

[30] Microsoft security bulletin. http://www.microsoft.com/technet/security/.

[31] Microsfot Security Intelligence Report, Volume 5, 2008. http://www.microsoft.com/security/portal/sir.aspx.

[32] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of Eurosys*, 2009.

[33] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[34] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[35] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, April 1986.

[36] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS TrustedWindow system. In *Usenix Security*, 2004.

[37] Symantec global internet security threat report: Trends for july - december 07, April 2008.

[38] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, October 2007.

[39] Cross-Domain Vulnerability In Microsoft Internet Explorer 6. `http://cyberinsecure.com/cross-domain-vulnerability-in-microsoft-internet-explorer-6/`.

[40] The XMLHttpRequest Object. `http://www.w3.org/TR/XMLHttpRequest/`.

[41] W3C XMLHttpRequest Level 2. `http://dev.w3.org/2006/webapi/XMLHttpRequest-2/`.

[42] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *To appear in the Proceedings of the IEEE Symposium on Security and Privacy*, May 2009.

[43] M. Zalewski. Browser security handbook, 2008. `http://code.google.com/p/browsersec/wiki/Main`.